# Appendix
# Learn2Assemble with Structured Representations and Search for Robotic Architectural Construction

**Niklas Funk, Georgia Chalvatzaki, Boris Belousov, and Jan Peters**
Department of Computer Science, Technical University of Darmstadt, Germany
{niklas,georgia,boris}@robot-learning.de, mail@jan-peters.net

In the following sections, we provide additional background information and implementation details concerning the methods and results presented in the paper. We start by describing the graph representations, before presenting the pseudocode for all learning algorithms and outlining the experimental setup in greater detail. Lastly, we provide hyperparameters, learning curves and additional material for all the experiments.

## A Graph representations

In this section, we introduce the Structure2Vec architecture, how to obtain every node's initial encoding and the two possibilities for defining the graph's connectivity. All these details are used in the experimental evaluation in Sec. 3.

### A.1 Alternative encoding - Structure2Vec (S2V)

**Structure2Vec Encoding (S2V).** S2V is one popular GNN architecture applied to combinatorial optimization algorithms [31]. In the first step of this method, the initial node embeddings $n_i^{(0)} = \mathbf{x}_i$ are projected into a higher dimensional space via

$$n_i^{(1)} = g(n_i^{(0)}) = \mathrm{ReLU}(\mathrm{FC}(n_i^{(0)})), \tag{1}$$

using a fully-connected (FC) layer followed by a rectified linear unit (ReLU) activation function. Note that the function $g$ will be used repeatedly as we progress with the graph update, however, on each further appearance we assume a different set of weights.

For discovering where a block needs to be placed next in our assembly problem, solely relying on the individual node encodings is not sufficient. We also need to take the topology of the graph into account. Thus, for each node, another feature $e_i$ is computed which accumulates the features of all its neighbors as follows

$$e_i(n_i)^{(1)} = g\left(\frac{1}{c({n_i}^{(0)})} \sum_{\mathcal{E}(i,j)=1, \forall j \in 1..N} g({n_j}^{(0)})\right), \tag{2}$$

where $c(n_i)$ is the connectivity count of the node $n_i$, indicating how many edges are connected to this node. Based on these two initial embeddings, for each node, we obtain high level features by performing $l$ rounds of message passing, i.e. updating each encoding $l$ times, according to

$$n_i^{(l)} = g\left({n_i}^{(l-1)}, g\left(e_i(n_i)^{(1)}, \frac{1}{c({n_i}^{(0)})} \sum_{\mathcal{E}(i,j)=1, \forall j \in 1..N} {n_j}^{(l-1)}\right)\right). \tag{3}$$

This high-level representation ${n_i}^{(l)}$ will be used subsequently to define our agent's action.

### A.2 Insights for the node encoding

To encode both the available and the already placed blocks, as well as the points representing the target structure to be built, we use the following initial representation

$$n_i^{(0)} = \begin{bmatrix} \mathbf{x}_{\text{pos}} \\ \text{placed} \\ \text{target} \end{bmatrix} \tag{4}$$

5th Conference on Robot Learning (CoRL 2021), London, UK.

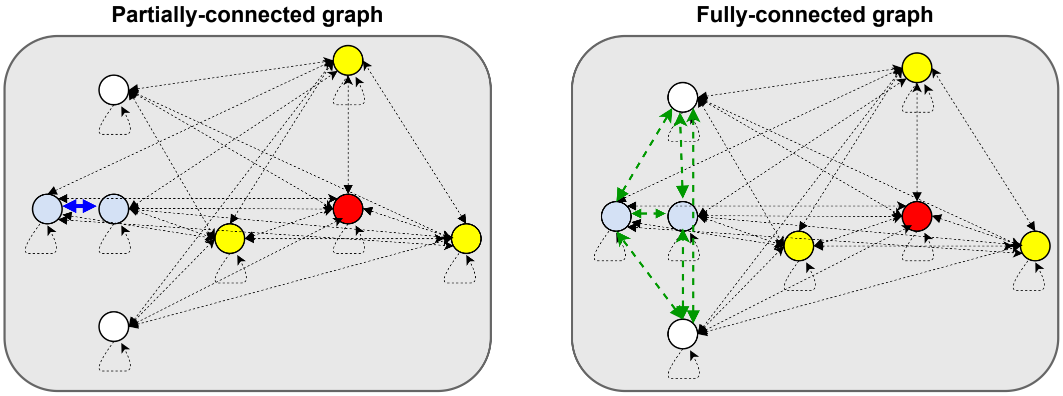


Figure 1: Illustrating the two different connectivity implementations. Note: The two nodes shaded in light blue illustrate one object made up of two primitive blocks. In the partial connectivity setup on the **left**, there are no connections in between the unplaced elements visualized on the left-hand side. Only the connection between the light blue nodes reveals them forming one object. On the contrary, in the fully-connected setup on the **right**, all nodes irrespective of their type are connected.

with the respective object's position ($\mathbf{x}_{\mathrm{pos}} \in \mathbb{R}^3$), and the booleans $\mathrm{placed}$ and $\mathrm{target}$. $\mathrm{placed}$ takes a value of 1 for target elements and already placed blocks, and -1 otherwise. $\mathrm{target}$ equates to 1 for target elements only, and to -1 otherwise. This initial representation forms the basis for obtaining the graph's higher-level encoding using either the MHA or the S2V architecture.

### A.3 Graph connectivity

In Fig. 1, we illustrate the different connectivities investigated in this work. Namely, the partially-connected and the fully-connected setup. Whereas in the fully-connected setup all nodes are simply interconnected with each other, in the partially-connected setup, there are no connections in between the unplaced objects. We reason that message passing between the unplaced objects can be omitted, as it should be more crucial to make the connections between the placed elements and the target shape in order to figure out where to next place an unused block. Further, the partial connectivity also has the advantage to clearly mark objects formed from multiple boxes, by adding connections between the individual primitive elements without requiring any further one-hot encoding or similar representation.

## B Learning algorithms - Pseudocode

This section provides the pseudocode of all the algorithms presented in Sec. 2.2. Alg. 1 represents the standard framework in which the learning is conducted, i.e. periodically collecting experience and updating the current Q-function which forms the basis for action selection. The difference between the algorithms lies in the action selection mechanisms and loss functions. Apart from that, all of them make use of a replay buffer during training and exploit the same simulation environments. For implementing the learning algorithms, we have used the mushroom reinforcement learning library [44].

The standard Q-learning algorithm, without conducting any model-based search, is provided in Alg. 2. Alg. 3 is an extension to Alg. 2 by adding tree search during test time. For expanding the initial node, it uses an epsilon-greedy expansion strategy. Algs. 5 and 4 differ from the others as also search is conducted while training the models. Whereas Q-MCTS conducts the tree search on every sample and uses an epsilon-greedy expansion, $\epsilon$-MCTS first has the decision whether to do search or not and then eventually uses a UCT expansion strategy. Thus, the loss function of $\epsilon$-MCTS is also more adaptive, since the cross-entropy loss term is only active when search has been conducted during training starting from the current state. As $\epsilon$ decreases over time, the influence of the cross-entropy term in $\epsilon$-MCTS is increasing. Contrarily, Q-MCTS does not have this decision and the cross-entropy term is active throughout the whole training process, as for every sample search is conducted.

**Algorithm 1** Learn2Assemble

```
 1: for $i$ in $1..$NumberEpochs do
 2: /* Collect experience
 3:     $j = 0$, Buffer $\mathcal{B} = []$
 4: /* Define number of samples to collect
 5:     $\Gamma = 100$
 6:     while $j < \Gamma$ do
 7:         Sample unplaced elements $\mathcal{S}_U$, initially placed box $\mathcal{S}_P$, target shape $\mathcal{S}_T$
 8:         finished=False
 9:         while finished==False do
10:             Sample action $[a, Q_S, a_e] = act(Q, s)$ using Q-function approximator $Q$
11:             Move robot to pick and place the part - Update: $\mathcal{S}_U$, $\mathcal{S}_P$, obtain $r(s, a)$
12:             Receive next state $s'$
13:             $\mathcal{B}.\text{append}([s, a, Q_S, a_e, r(s, a), s'])$
14:             $j = j + 1$
15:             $s = s'$
16:             if $|\mathcal{S}_U| = 0$, or robot destroyed the structure, or $F(s_{t+1}) > \Delta$, or $j == \Gamma$ then
17:                 finished=True
18: /* Update weights of Q-function
19:     $\pi = update(Q, \mathcal{B})$
```

**Algorithm 2** DQN

```
 1: Number of update steps $\chi$
 2: procedure $act(Q, s)$
 3:     if $\text{RandomVariable} < \epsilon$ then
 4:         $a = \text{RandomChoice}(\text{AllPossibleActions(s)})$
 5:     else
 6:         $a = \max_{a'} Q(s, a'|a' \in \text{AllPossibleActions}(s))$
        return $a$
 7: procedure $update(\pi, \mathcal{B})$
 8:     Add $\mathcal{B}$ to Replay Memory
 9:     for $i$ in $1..\chi$ do
10:         Sample random subset from Replay Memory
11:         $\text{loss} = \text{smoothL1}(Q(s, a) - (r(s, a) + \gamma \max_{a'} Q_T(s', a'|a' \in \text{AllPossibleActions}(s))))$
12:         Update Q-function approximator $Q$ with parameters $\theta$
13:         $\theta = \theta - \alpha \frac{\partial \text{loss}}{\partial \theta}$
14:     return $Q$
```

**Algorithm 3** DQN + MCTS

```
 1: /* Note, this is only during evaluation, for training see Alg. 2
 2: Rollout Depth $\eta = 1$ if not stated otherwise
 3: Search Budget $\tau$
 4: Number of update steps $\chi$
 5: procedure $act(Q, s)$
 6:     Given: state $s$, set containing the explored actions $\mathcal{S}_A = \{\}$
 7:     $\forall a$, Initialize $W(s, a) = 1$, $Q_S(s, a) = Q(s, a)$
 8:     for $i$ in $1..\tau$ do
 9:         if $\text{RandomVariable} < \epsilon$ then
10:             $a = \text{RandomChoice}(\text{AllPossibleActions(s)}|W(s, a) = 1)$
11:         else
12:             $a = \max_{a'} Q(s, a'|a' \in \text{AllPossibleActions}(s), W(s, a') = 1)$
13:         Add $a$ to $\mathcal{S}_A$, collect $r(s, a)$
14:         for $j$ in $1..\eta - 1$ do
15:             $a = \text{DQN} - \text{act}(Q, s)$ (Alg. 2, Line 2), collect current single step reward $\tilde{r}$
16:             Update: $r(s, a) = r(s, a) + \gamma^j \tilde{r}$
17:         Update: $W(s, a) = W(s, a) + 1$, $Q_S(s, a) = \frac{1}{2}(Q_S(s, a) + r(s, a) + \gamma^\eta \max_{a'} Q(s', a'))$
18:     if $\text{RandomVariable} < \epsilon$ then
19:         $a_r = \text{RandomChoice}(\mathcal{S}_\text{A})$
20:     else
21:         $a_r = \max_{a'} Q_S(s, a'|a' \in \mathcal{S}_A)$
22:     return $a_r, \{Q_S(s, a|a \in \mathcal{S}_A)\}, \{a|a \in \mathcal{S}_A\}$
```

**Algorithm 4** Q-MCTS

```
Rollout Depth $\eta = 1$ if not stated otherwise
Search Budget $\tau$
Number of update steps $\chi$
procedure $act(Q, s)$
    Given: state $s$, set containing the explored actions $\mathcal{S}_A = \{\}$
    $\forall a$, Initialize $W(s,a) = 1$, $Q_S(s,a) = Q(s,a)$
    for $i$ in $1..\tau$ do
        if $\mathrm{RandomVariable} < \epsilon$ then
            $a = \mathrm{RandomChoice}(\mathrm{AllPossibleActions(s)} | W(s,a) = 1)$
        else
            $a = \max_{a'} Q(s, a' | a' \in \mathrm{AllPossibleActions}(s), W(s,a') = 1)$
        Add $a$ to $\mathcal{S}_A$, collect $r(s,a)$
        for $j$ in $1..\eta - 1$ do
            $a = \mathrm{DQN} - \mathrm{act}(Q, s)$ (Alg. 2, Line 2), collect current single step reward $\tilde{r}$
            Update: $r(s,a) = r(s,a) + \gamma^j \tilde{r}$
        Update: $W(s,a) = W(s,a) + 1$, $Q_S(s,a) = \frac{1}{2}(Q_S(s,a) + r(s,a) + \gamma^\eta \max_{a'} Q(s',a'))$
    if $\mathrm{RandomVariable} < \epsilon$ then
        $a_r = \mathrm{RandomChoice}(\mathcal{S}_\mathrm{A})$
    else
        $a_r = \max_{a'} Q_S(s, a' | a' \in \mathcal{S}_A)$
    return $a_r, \{Q_S(s, a | a \in \mathcal{S}_A)\}, \{a | a \in \mathcal{S}_A\}$
procedure $update(\pi, \mathcal{B})$
    Add $\mathcal{B}$ to Replay Memory
    for $i$ in $1..\chi$ do
        Sample random subset from Replay Memory, including the actions taken during search
        loss $= \frac{1}{2}\mathrm{smoothL1}(Q(s,a) - (r(s,a) + \gamma \max_{a'} Q_T(s', a' | a' \in \mathrm{AllPossibleActions}(s))))$
            $- \frac{1}{2}\mathrm{softmax}(Q(s, a_e))^T \mathrm{softmax}(Q_S(s, a_e))$
        Update Q-function approximator $Q$ with parameters $\theta$
        $\theta = \theta - \alpha \frac{\partial \mathrm{loss}}{\partial \theta}$
    return $Q$
```

# C Additional details on experimental setup

All the simulation environments have been implemented using PyBullet [45]. Every environment is characterized by the size and number of all the target shapes, as well as the number of blocks that are available. As most of the components used for the learning algorithms are implemented with fixed-size arrays, we decided that during training, every graph has a fixed size, i.e. a fixed number of nodes. However, as per side, the number of points describing the target shape is variable (3, 4 or 5), the number of available blocks is adapted accordingly. Thus, despite the choice of fixing the number of nodes during training, the algorithms still encounters a versatile set of graphs as the distribution over the node types changes depending on how many nodes are required for describing the desired shape.

## C.1 Sampling target shape

For every side of the environment, we initially define its size. In this work, we experimented with grids ranging from size 3-by-3 up to sizes of 6-by-6. Exemplarily, a grid sized 3-by-3 has a maximum width of 3 blocks and a maximum height of 3 blocks. We assume to place the initial element (red block) always in the bottom center of this grid. If there are multiple sides to be built, the initial block is placed by randomly selecting one of them. From inside these grids, we sample 3, 4 or 5 points. Two out of these points are always sampled on the bottom, one to the left of the initial block and one to the right of it. For the upper points, we just sample points at a random height and width from inside the available grid.

**Algorithm 5** $\epsilon$-MCTS

```
Rollout Depth $\eta = 1$ if not stated otherwise
Search Budget $\tau$
Number of update steps $\chi$
procedure $act(Q, s)$
    if $\text{RandomVariable} < \epsilon$ then
        $a = \text{RandomChoice}(\text{AllPossibleActions(s)})$
        return $a$
    else
        Given: state $s$, set containing the explored actions $\mathcal{S}_A = \{\}$
        $\forall a$, Initialize $W(s, a) = 1$, $Q_S(s, a) = Q(s, a)$
        for $i$ in $1..\tau$ do
            $a = \max_{a'} Q(s, a') + 2\sqrt{\frac{log(\sum_{a''} W(s,a''))}{W(s,a')}} | a' \in \text{AllPossibleActions}(s), W(s, a') = 1$
            Add $a$ to $\mathcal{S}_A$, collect $r(s, a)$
            for $j$ in $1..\eta - 1$ do
                $a = \text{DQN} - \text{act}(\pi, s)$ (Alg. 2, Line 2), collect current single step reward $\tilde{r}$
                Update: $r(s, a) = r(s, a) + \gamma^j \tilde{r}$
            Update: $W(s, a) = W(s, a) + 1$, $Q_S(s, a) = \frac{1}{2}(Q_S(s, a) + r(s, a) + \gamma^\eta \max_{a'} Q(s', a'))$
        $a_r = \max_{a'} Q_S(s, a' | a' \in \mathcal{S}_A)$
        return $a_r, \{Q_S(s, a | a \in \mathcal{S}_A)\}, \{a | a \in \mathcal{S}_A\}$
procedure $update(Q, \mathcal{B})$
    Add $\mathcal{B}$ to Replay Memory
    for $i$ in $1..\chi$ do
        Sample random subset from Replay Memory, including the actions taken during search
/* Note: If the sample is collected without conducting search, then, for this sample, the cross entropy
regularization term is omitted and instead equates to 0.
        loss $= \frac{1}{2}\text{smoothL1}(Q(s, a) - (r(s, a) + \gamma \max_{a'} Q_T(s', a' | a' \in \text{AllPossibleActions}(s))))$
            $- \frac{1}{2}\text{softmax}(Q(s, a_e))^T \text{softmax}(Q_S(s, a_e))$
        Update Q-function approximator $Q$ with parameters $\theta$
        $\theta = \theta - \alpha \frac{\partial \text{loss}}{\partial \theta}$
    return $Q$
```

### C.2 Reward and reset function

Our method's objective is to use the available blocks to construct a stable structure that fills the target design up to a desired filling threshold $\Delta$ for which we view the experiment as successful. This results in a reset of the environment, i.e., sampling a new instance with new target points and blocks. The environment is also reset when there are no more unplaced blocks available, or if an invalid action has been taken. An action is invalid if it results in an unstable configuration, in destructing the previously built structure, or if it is not executable by the robot due to kinematic constraints. The outcome of every action is checked through the simulator as explained in Sec. C.5. To refine the stacking policies, we therefore have to provide a meaningful reward signal to incentivize the successful completion of the construction tasks. We have used two different reward definitions in this work, depending on the simulation environment. In the simple block stacking environments with only one type of block available, we made use of a discrete reward as introduced in Sec. C.2.1. However, in the more complicated environments with multiple different blocks available, we defined another reward function (see Sec. C.2.2) which not only returns a binary signal but actually encourages to fill the shape as efficiently as possible by providing a reward proportional to the increase in the filling.

#### C.2.1 Reward function in simple block stacking environments

In the simple environments with only one type of blocks available, we define the reward as

$$r(s_t, a_t) = \begin{cases} 1 & \text{if } F(s_{t+1}) - F(s_t) > 0, \\ -1 & \text{if an invalid action,} \\ 0 & \text{otherwise,} \end{cases} \tag{5}$$

where an invalid action is defined as described above in Sec. C.2. The coverage $F(s_t)$ is computed by first summing over the intersection areas for each side, i.e. the areas for which the target shape and current structure overlap, and divide it by the total area of the target shapes. The intuition behind this reward function is to provide an "intrinsic" motivation for actions that lead to improvement in the filling of the desired area, punish actions that disrupt the execution, and to not reward any actions that do not promote the assembly (e.g., placing blocks outside the desired area).

### C.2.2 Reward function in environments with multiple different blocks

In environments with multiple different blocks available we want to incentivize filling the target shape using the different available modules as effectively as possible, and thus assign a slightly modified reward function according to

$$r(s_t, a_t) = \begin{cases} c_1(F(s_{t+1}) - F(s_t)) + 1 & \text{if } F(s_{t+1}) - F(s_t) > 0 \text{ and } F(s_{t+1}) > \Delta, \\ c_1(F(s_{t+1}) - F(s_t)) & \text{if } F(s_{t+1}) - F(s_t) > 0, \\ -1 & \text{if an invalid action destroying the structure,} \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

The coverage $F(s_t)$ is computed as introduced in Sec. C.2.1 and the hyperparameter (scaling constant) $c_1$ is set to 3. The constant has been determined empirically and is required to scale the reward signal to facilitate the learning of the action-value function. For example, setting this constant to 1 might make it difficult to differentiate an action that results in only marginally improving the filing from another one that does not improve it at all. Since exceeding the filling threshold $\Delta$ corresponds with successfully completing the experiment, we provide an additional bonus of +1 to the agent.

## C.3 Populate environment without (wo) robot

Depending on the choice of target shape, as well as the number of available blocks, we have $N_k$ blocks remaining that need to be placed. To ensure an equal spacing in between the blocks, we use Sobol sequences [46] to sample where the unplaced blocks are to be placed. Note that in the experiments without the robot, we did not enforce any measures to avoid collisions between the unplaced blocks, as they are not modelled by the physical simulator. They are only added to the simulated scene the moment they are placed.

## C.4 Populate environment with (w) robot

In the environment with the robot, the unplaced parts are added to the physics simulator from the very beginning, as the parts have to be grasped by the robot. To ensure graspability, we place them in rows with sufficient spacing.

## C.5 Checking stability of overall structure

To check the stability of the overall structure, we track the velocities of all the parts in the scene. If blocks are colliding or falling down, the accumulated velocity will exceed a threshold, which will signal that the action has been invalid. This will reset the entire environment. In all non-robotic scenarios, this threshold on the velocity is also active for the block that is being placed, ensuring that the controller will not drop any block from above, as this will result in more inaccurate placements compared to just positioning it exactly at the right place. In the robot scenarios, and especially in the multidimensional ones, dropping a cube from a higher position is sometimes required as otherwise, the construction of enclosed shapes is impossible. Therefore, we only keep track of all the placed elements, ignoring the block that is currently being placed.

## C.6 Robotic environment - Moving the robot

For moving the robot, we use trajectories that are defined by multiple waypoints. To map from the Cartesian space to joint coordinates, we use the Pinocchio library [47]. For grasping as well as placing the cubes, the respective goal positions are approached from the top, without running any additional obstacle avoidance module. As shown in Fig. 2, the objects are grasped from the top and there are two grasping poses, as well as two placing poses available. These four grasp-place

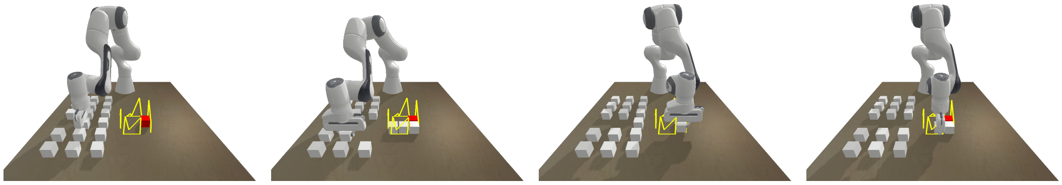

(a) (b) (c) (d)

Figure 2: Illustrating the different available grasping and placing poses in the four-sided robotic environment. **(a)&(b)** Two grasping poses. **(b)&(c)** Two placing poses.

Table 1: Hyperparameters used for training the policies

| Parameter | Value |
|---|---|
| Number Training Epochs | 5000 |
| Discount Factor $\gamma$ | 0.8 |
| Initial $\epsilon$ | 1.0 |
| Final & evaluation $\epsilon$ | 0.05 |
| Epoch final $\epsilon$ reached | 1000 |
| Size replay buffer | 30000 |
| Optimizer | adam |
| Learning rate | 0.001 |
| Batch size | 32 |
| Number of update steps $\chi$ | 25 |
| Update target (epochs) | 50 |
| Target filling threshold $\Delta$ | 0.975 |

combinations allow to realize rotations by $0°$ (same orientation for grasping and placing), as well as $\pm 90°$ around the z-axis.

## D Additional experimental results

In this section, we will provide additional details and results underlining the main findings of our work. To better showcase the behavior of the individual agents, we also provide videos on our website `https://sites.google.com/view/learn2assemble`.

### D.1 General information on training procedure

If not stated otherwise, all the results presented in the paper are based on evaluating 5 agents that have been trained with different random seeds using the parameters shown in Table 1.

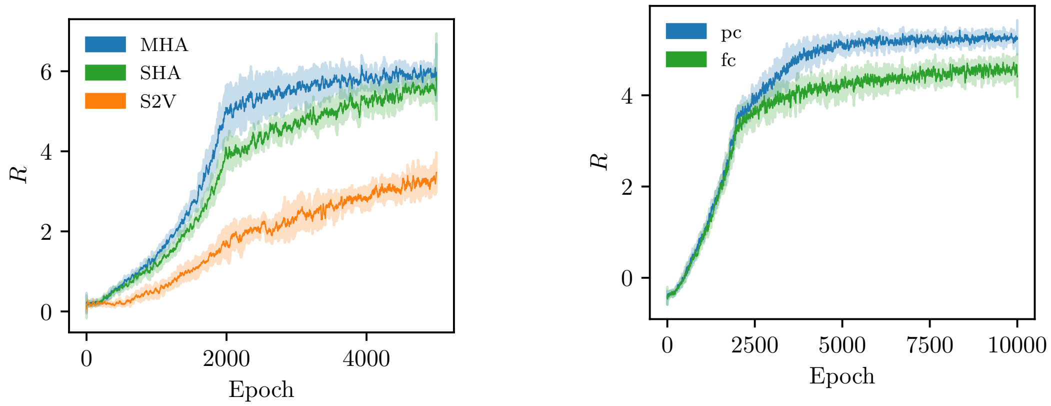
For all experimental results presented in this work, we report the mean values (and eventually the 95% confidence interval in brackets) from evaluating all the agents on building 100 randomly sampled target shapes. In the result tables in Sec.3, the star(*) indicates the environment in which the agents have been trained in, while the other experiments are OOD.

The value of $\epsilon$ is decreased linearly and the target Q-network $Q_T$ is updated every 50 epochs, where each epoch consists of sampling 100 state-action transitions. To speed up the training process, which is especially crucial when search is added also during training time, we implemented a parallel sampling strategy by combining the code from [48] and [44]. In the following, we will also show the evolution of the discounted average return during the training process. Note that for obtaining those learning curves, we did not average over building 100 target shapes, but use the score collected from the 100 samples which are at the same time exploited to update the Q-function.

### D.2 Evaluation of graph architectures

For evaluating the graph architectures, we used a slightly different set of parameters. Namely, the final exploration frame has been reached after episode 2000 and the replay buffer size has been set to 150000.

As illustrated in Fig. 3a, already throughout the training process, the discounted average return obtained by the MHA agents is significantly higher compared to using the S2V architecture. This hints that using the attention mechanism results in obtaining more powerful representations, which then result in obtaining higher rewards as the target shape can be filled more effectively. The videos



(a) Evolution of the discounted average return $R$ when training the MHA, SHA and S2V models using the standard DQN algorithm (Alg. 2).

(b) Evolution of the discounted average return $R$ when training the MHA in the fully-connected (fc) and partially-connected (pc) configuration using the standard DQN algorithm (Alg. 2).

Figure 3: Learning curves for different experiments.

Table 2: Comparison of different architectures on the single-sided environment without the robot and only one type of block, extending the results presented in Table 1 by also reporting scores for the single-head attention encoding (SHA). $R$ is the cumulative discounted return, $f$ the ratio of runs that ended with failure, i.e., the structure colliding and $b$, the ratio of runs that ended without success and no more blocks remaining.

| | 3-by-3 grid, 20-24 blocks* | | | 3-by-3 grid, 30-34 blocks | | | 4-by-4 grid, 20-24 blocks | | |
|---|---|---|---|---|---|---|---|---|---|
| Method | $R$ | $b$ | $f$ | $R$ | $b$ | $f$ | $R$ | $b$ | $f$ |
| MHA (FC) | **3.22** (0.04) | 0.01 | **0.15** | **3.44** (0.04) | 0.00 | **0.21** | **3.66** (0.05) | 0.18 | **0.41** |
| SHA (FC) | 2.99 (0.09) | 0.08 | 0.25 | 3.16 (0.10) | 0.04 | 0.37 | 3.48 (0.08) | 0.25 | 0.49 |
| S2V (FC) | 2.36 (0.08) | 0.49 | 0.47 | 2.15 (0.10) | 0.08 | 0.87 | 2.75 (0.20) | 0.45 | 0.55 |

confirm these impressions and clearly show that the S2V architecture has difficulties drawing the connection between the target shape and the already placed blocks, resulting in placing many unnecessary blocks and failing to fill the target shape reliably. This underlines the findings presented in Sec. 3.

### D.2.1 Additional experiment - Single-head attention (SHA)

To evaluate the effectiveness of using the multi-head attention approach, we ran one additional experiment using only a single attention head. The results are shown in Table 2. We reason that SHA performs better compared to S2V as it can explicitly compute the compatibility score between nodes which helps to disambiguate the different components encoded in the graph structure. Nevertheless, the superiority of MHA hints that having multiple attention heads with different weights allows to construct even more meaningful features which result in better performance.

## D.3 Evaluation of graph connectivity

While it is quite straightforward to defined each node's features (position, type information, cf. Appx. A.2), there are different approaches for the graph connectivity. This choice is important as it influences the message passing between nodes. We investigated the effect of using a fully connected (FC) graph and compare it to using partial connections (see Fig. 1) omitting the connections between different unplaced blocks, inducing a stronger inductive bias on the structured representation.

To test the generalization abilities, we test the approaches in the setting of Fig. 4 with different types of blocks (i.e., rectangles of various lengths). Those novel objects are fixed concatenations of the already existing blocks, treating the individual boxes as primitive elements. This has the advantage of not needing any modifications to the action and observation space, while allowing us to create more complex and diverse objects.

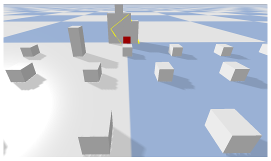


Figure 4

Table 3: Comparison of FC against partially connected MHA architecture on a task with variable sets of object-types available in the setting shown in Fig. 4.

| Method | 5-by-5 grid, 25-27 blocks * |  | 5-by-5 grid, 35-37 blocks |  | 6-by-6 grid, 35-37 blocks |  | 5-by-5 grid, 25-27 blocks (different types) |  |
|---|---|---|---|---|---|---|---|---|
|  | $R$ | $f$ | $R$ | $f$ | $R$ | $f$ | $R$ | $f$ |
| MHA (FC) | 1.94 (0.12) | 0.39 | 1.69 (0.11) | 0.51 | 1.32 (0.10) | 0.61 | 1.88 (0.11) | 0.42 |
| MHA (partial) | **2.42** (0.03) | **0.23** | **2.22** (0.06) | **0.34** | **1.75** (0.08) | **0.51** | **2.40** (0.10) | **0.26** |

We use the same set of parameters as in the previous subsection, but we trained the agents for 10000 epochs. Further, as this experiment considers also larger blocks, we use the adapted reward function introduced in Appx. C.2.2. Using the larger objects necessitates two new skills: learning to differentiate the different object types and handling them correctly.

*Results.* As shown in Table 3, partial connectivity outperforms the FC in all experiments. Interestingly, designing a priori meaningful edge connections favors the extrapolation of the method to unseen types of objects (last column), indicating that the chosen representation is modular and effective in solving more challenging problem settings. These results also indicate that reasoning about different block types in the FC setting is more difficult, as this has to happen based on the blocks' distances. Contrarily, in the partial setting, the connectivity information directly reveals the block type. As we want to keep the flexibility of handling more complex blocks, we will exclusively focus on the partially connected architecture.

Fig. 3b depicts the learning behaviors for the fully- and the partially-connected setup. While initially, there is hardly any difference between the two approaches, in the long run, the partially-connected setup clearly outperforms the fully-connected one. We reason that during the early stages of training, the connectivity only plays a minor role, as all the agents have to figure out which actions are admissible. However, as training proceeds, the partial connectivity gathers higher rewards, indicating that providing additional structure through the graph's connectivity facilitates learning. This trend is also visible in the accompanying videos.

### D.4 Evaluation of the learning algorithms

When comparing the performance of the different learning algorithms, we use the exact same hyperparameters as introduced in Sec. D.1. Compared to the experiments presented in Secs. D.2 & D.3, we decided to decrease the size of the replay buffer and to increase the speed of decaying $\epsilon$. Those measures were intended to increase memory efficiency as well as to improve the convergence speed. Further, for training the Q-MCTS and the $\epsilon$-MCTS agents a search budget of 5 is being used.

The learning progress is very similar for all the introduced algorithms, as can be seen in Fig. 5. The learning curves for DQN as well as $\epsilon$-MCTS are hardly distinguishable, which we think is due to the fact that both of them are using an $\epsilon$-greedy decision whether to apply a random action or to either use the best action according to the Q-function (DQN) or according to the result of a quick searching procedure ($\epsilon$-MCTS). Q-MCTS, in contrast, seems to learn slightly quicker during the early phase of the learning process due to more exploitation, as it always performs the tree search. However, this behavior might actually also result in less exploration, as always conducting tree search might result in overfitting to a suboptimal solution.

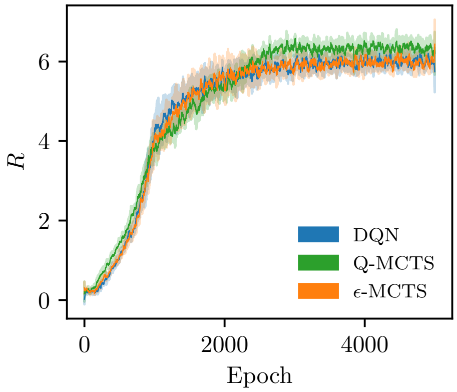


Figure 5: Evolution of the discounted average return $R$ when training agents using the different learning algorithms.

Table 4 provides results from running additional experiments. The last two columns (most difficult tasks) reveal that using a search budget of 10, instead of 5, slightly improves performance. While using a search budget of 5 and playing the rollout until termination yields best performance across all tasks and methods, we still decided to use a search budget of 10 with only a single step of expansion for the subsequent experiments. The main reason for this choice being computational efficiency.

Table 4: Combining Q-learning and MCTS in the two-sided environment without the robot. $R$ denotes the cumulative discounted return, $\bar{a}$, the mean number of actions conducted in this environment, $f$ the ratio of runs that ended with failure, i.e. the structure colliding. The double star (**) indicates that the rollout has been played until termination.

| Search | | 3-by-3, grid 20-24 blocks* | | | 3-by-3 grid, 30-34 blocks | | | 4-by-4 grid, 30-34 blocks | | | 5-by-5 grid, 40-44 blocks | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Budget | Method | $R$ | $\bar{a}$ | $f$ | $R$ | $\bar{a}$ | $f$ | $R$ | $\bar{a}$ | $f$ | $R$ | $\bar{a}$ | $f$ |
| 0 | DQN | **3.21** (0.05) | 7.93 | 0.16 | **3.44** (0.08) | 8.65 | 0.17 | **3.61** (0.06) | 12.02 | 0.51 | **3.63** (0.08) | 13.66 | 0.93 |
| | $\epsilon$-MCTS | 3.18 (0.03) | 8.57 | 0.22 | 3.37 (0.10) | 9.30 | 0.30 | 3.54 (0.09) | 12.53 | 0.62 | 3.50 (0.13) | 12.75 | 0.95 |
| | Q-MCTS | 2.80 (0.15) | 7.33 | 0.51 | 2.89 (0.09) | 7.60 | 0.64 | 2.66 (0.08) | 7.47 | 0.92 | 2.34 (0.17) | 6.45 | 0.99 |
| 5 | DQN+MCTS | **3.43** (0.04) | 8.20 | 0.03 | **3.67** (0.02) | 8.99 | **0.02** | **3.95** (0.04) | 14.05 | 0.35 | **3.96** (0.10) | 16.05 | 0.91 |
| | $\epsilon$-MCTS | 3.16 (0.06) | 8.38 | 0.23 | 3.34 (0.08) | 9.11 | 0.35 | 3.55 (0.09) | 12.45 | 0.61 | 3.51 (0.18) | 12.61 | 0.93 |
| | Q-MCTS | 3.07 (0.10) | 8.03 | 0.40 | 3.20 (0.08) | 8.35 | 0.50 | 3.10 (0.12) | 8.97 | 0.85 | 2.85 (0.19) | 8.27 | 0.98 |
| 5** | DQN+MCTS | **3.47** (0.02) | 8.12 | **0.02** | **3.71** (0.02) | 8.83 | **0.02** | **3.99** (0.04) | 14.13 | **0.28** | **4.04** (0.06) | 17.08 | **0.87** |
| | $\epsilon$-MCTS | 3.21 (0.04) | 8.40 | 0.19 | 3.39 (0.06) | 9.06 | 0.31 | 3.60 (0.06) | 12.84 | 0.59 | 3.58 (0.12) | 13.28 | 0.92 |
| | Q-MCTS | 3.20 (0.07) | 8.00 | 0.27 | 3.28 (0.10) | 8.48 | 0.43 | 3.28 (0.07) | 9.60 | 0.77 | 3.08 (0.18) | 9.23 | 0.98 |
| 10 | DQN+MCTS | **3.47** (0.02) | 8.16 | 0.05 | **3.66** (0.03) | 8.96 | 0.06 | **3.96** (0.04) | 13.92 | 0.31 | **4.08** (0.09) | 17.33 | **0.87** |
| | $\epsilon$-MCTS | 3.21 (0.03) | 8.48 | 0.20 | 3.37 (0.11) | 9.10 | 0.31 | 3.60 (0.09) | 12.65 | 0.61 | 3.63 (0.16) | 13.16 | 0.93 |
| | Q-MCTS | 3.24 (0.04) | 8.56 | 0.27 | 3.39 (0.08) | 9.24 | 0.41 | 3.42 (0.07) | 10.99 | 0.76 | 3.41 (0.13) | 11.27 | 0.96 |
| 1000 | UCT | 0.54 | 4.00 | 1.00 | - | - | - | - | - | - | - | - | - |

Table 5: Comparing policies trained with (w) and without (wo) the robot-in-the-loop.

| | 5 by 5 grid 15-17 blocks* | | | 6 by 6 grid 16-18 blocks | | |
|---|---|---|---|---|---|---|
| Method | $R$ | $\bar{a}$ | $f$ | $R$ | $\bar{a}$ | $f$ |
| w robot | **2.71** (0.14) | 5.40 | **0.25** | **2.93** (0.08) | 6.71 | **0.42** |
| wo robot | 2.14 (0.27) | 3.92 | 0.40 | 2.40 (0.18) | 4.91 | 0.58 |

The video material that we provide along this appendix illustrates the different findings, as well as the results from Sec. 3. Adding search to the DQN agent improves performance and results in slight gains compared to the $\epsilon$-MCTS agents, whereas the Q-MCTS agents perform worse.

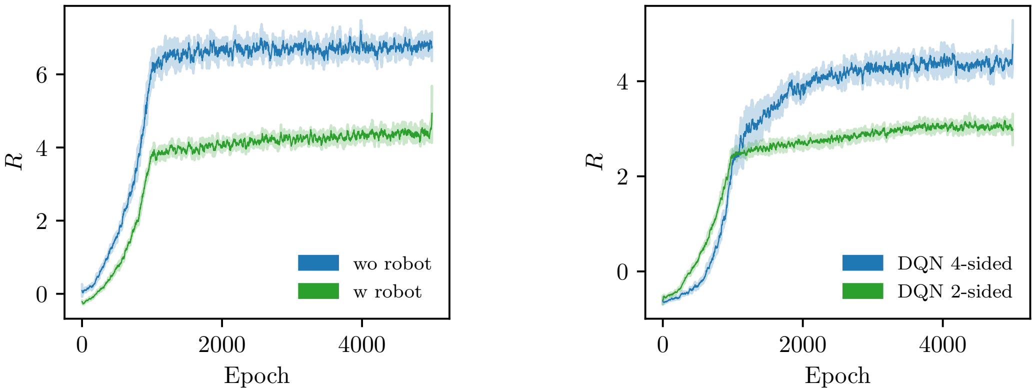
### D.5 Evaluation of trainings with and without the robot-in-the-loop

In this section, we provide more details on the comparison between DQN agents trained with and without the robot-in-the-loop. Compared to the previous experiments, we decided to adapt the filling threshold indicating when to view a shape as being built successfully. For all experiments including the robot, we decided to lower this value to $\Delta = 0.75$. The reason lies in the fact that when using the robot to actually place the cubes, there will always remain some spacing in between the parts. Therefore, lowering the threshold is necessary.

The difference in the magnitude of the obtained rewards in Fig. 6a can thus be explained by the two different threshold values used during training, as for the training without the robot, we still used the previous value of $0.975$. Apart from this, there does not seem to be a distinction in the learning speed. However, when evaluating both of the policies in the environment with the robot-in-the-loop, agents trained without the robot fail significantly more often as they do not take the robot's kinematics into account (see Table 5). This results in the arm colliding with the structure as shown in the accompanying videos. Thus, as already pointed out in Sec. 3 of the main paper, for obtaining reliable policies for assembling architectural structures, it is essential to include the robot-in-the-loop already during training. The results also underline that the graph representations are flexible enough to take the robot's restrictions and nonlinearities into account while still successfully solving the tasks. Note that for this experiment we still considered the 5-dimensional action space (only consisting of the placement actions) as choosing a grasp perpendicular to the structure being built will ensure to not collide with it. This choice also ensures a fair comparison between the agents.

### D.6 Evaluation of building complex shapes with the robot-in-the-loop

Lastly, we investigated building even more complex shapes with the robot-in-the-loop. Building those two- and four-sided shapes also requires grasp selection, and thus this setting is the most complex investigated in this work. Fig. 6b depicts the learning curves for the DQN agents trained on these difficult tasks. As can be seen, learning to build two-sided shapes seems to be more straightforward and results in faster training and a steeper learning curve. Nevertheless, towards the end of the training, the agents can achieve higher rewards in the four-sided environments as those contain more blocks to be placed. Since adding MCTS during test time improved performance throughout all of our experiments, we showcase the behavior of DQN+MCTS agents in the accompanying videos. Note that to better illustrate the performance of the agents, the threshold has been set to 0.9 for the



(a) Evolution of the discounted average return $R$ when training the agents on the single-sided environment with and without the robot-in-the-loop using the standard DQN algorithm (Alg. 2).

(b) Evolution of the discounted average return $R$ when training the agents in the two- and four-sided environment, including the robot using the standard DQN algorithm (Alg. 2).

Figure 6: Learning curves for different experiments with the robot-in-the-loop.

videos. As shown there, the agents are capable of building complex shapes and are successful at placing a large number of blocks.

### D.7 Illustrating search

We present a series of pictures, which depict the searching process, i.e. which actions have been explored during search given the initial configuration shown in Fig. 7. As can be seen in Figs. 8 and 9 which show the explored grasping and placement actions respectively, the tree search explores a versatile set of actions and attempts to grasp different objects. Some of the grasps are invalid as the gripper fails to grasp the part due to collisions (see Fig. 8 a). The other grasps are valid but the associated placement actions illustrated in Fig. 9 b-d result in a different filling of the target structure. Finally, the highest reward action (grasp - Fig. 8 d, placement - Fig. 9 d)is selected and executed in the environment.

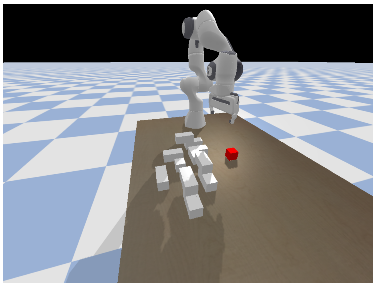
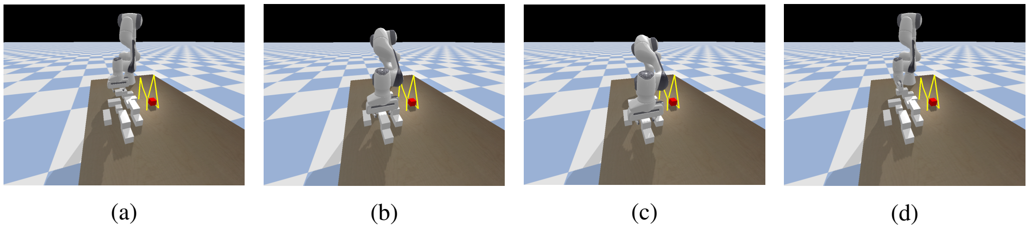
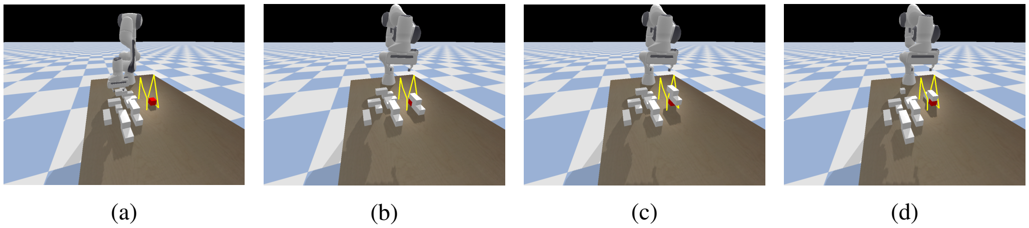


Figure 7: Initial configuration on which we start the search.



(a) (b) (c) (d)

Figure 8: **(a) - (d)** Illustrating four different grasping configurations explored during search. Only the grasp depicted in **(a)** is invalid as the gripper collides with the block.



(a) (b) (c) (d)

Figure 9: **(a) - (d)** Illustrating the result of executing the placement actions (including gripper orientation) starting from the grasps illustrated in Fig. 8. For **(a)** no placement is executed as the grasp is already invalid. Given the target shape, the placement action depicted in **(d)** results in the highest reward and is thus finally executed after the search.

### D.8 Generalization with respect to randomized scenes

For the experiments on how well our learned representations scale to different scenes, we have used the exact same training configuration as for the results in Appx. D.6. In fact, we have simply evaluated the agents that have been trained in the original two-sided scenario and evaluate them in the randomized scenes. As described in the main paper, and as also shown in the accompanying videos (filling threshold set to 0.9), the agents transfer well to the novel settings.

### D.9 Generalization with respect to different building blocks

Figure 10 illustrates the environment in which multiple blocks are available. As shown, there are in total 4 different building blocks available which are i) primitive box object (from previous experiments), ii) vertical block of length 2, consisting of 2 primitive blocks, iii) L-shaped object consisting of 3 primitive blocks, and iv) s-shaped object which is made up of 4 primitive blocks. In line with the previous experiments, the set of objects that is available is sampled at random from this set. This however implies that now there is no guarantee that the set of available blocks is actually sufficient to fill the shape up to the desired threshold. Also, for the objects that are not symmetric around the z-axis, i.e. L- and s-shaped block, we randomly choose an orientation from the set $\{0, \pm 90, 180\}$.

In the environments without the robot, we modify the action space, as it would otherwise be impossible for the agents to rotate the available blocks which might be crucial to successfully solve these more complicated tasks. We therefore also allow the agents to rotate the respective block around the z-axis with values of $\{0, \pm 90, 180\}$. The combination of those four rotational actions together with the five placement actions results in an action space of dimension $N_a = 4 \times 5 = 20$.

For training the agents in those complicated settings we use the reward function from Appx. C.2.2. We also found that applying the previous settings (cf. Table 1) results in converging to suboptimal

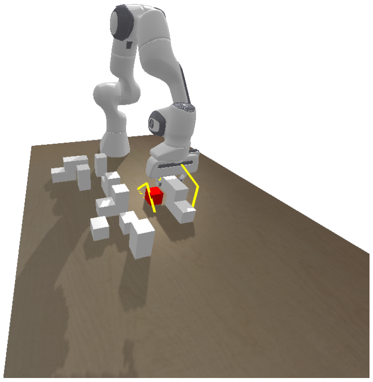


Figure 10: Initial configuration on which we start the search.

Table 6: Evaluating the trained policies in the environments with multiple different objects available. The results in row 3&4 correspond to using a modified environment without including the robot.

| | 5 by 5 grid 20-24 blocks* | | | 5 by 5 grid 30-34 blocks | | | 6 by 6 grid 30-34 blocks | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Method | $R$ | $\bar{a}$ | $f$ | $R$ | $\bar{a}$ | $f$ | $R$ | $\bar{a}$ | $f$ | w/wo robot |
| DQN | 1.25 (0.03) | 3.18 | 0.25 | 0.28 (0.07) | 2.95 | 0.67 | 0.11 (0.11) | 3.12 | 0.75 | w robot |
| DQN+MCTS | 1.42 (0.05) | 3.53 | **0.21** | 0.77 (0.09) | 3.27 | **0.52** | 0.61 (0.08) | 4.38 | **0.61** | w robot |
| DQN | 1.44 (0.21) | 4.87 | 0.34 | 1.29 (0.13) | 5.91 | 0.43 | 0.94 (0.15) | 6.53 | 0.56 | wo robot |
| DQN+MCTS | **2.20** (0.02) | 5.05 | **0.06** | 1.89 (0.03) | 5.96 | **0.15** | 1.48 (0.09) | 7.27 | **0.27** | wo robot |

solutions. We thus increased the number of training epochs from 5000 to 10000, increased the epoch when the final value of $\epsilon$ is reached from 1000 to 2000, and also increased the discount factor to $0.95$.

Table 6 complements the results in the paper by also reporting the performance of the DQN agents. As can be seen, also in those scenarios, the addition of search improves the agent's performance. However, in the experiments that include the robot, the addition of search is not as efficient as for the other experiments (especially considering the first column) which hints at the fact that there might be additional limiting factors. We reason that the action space might limit the agent's performance by not providing enough flexibility in terms of low-level placement actions, as well as grasp selection.